Fortress Programming Language Tutorial

Guy Steele and Jan-Willem Maessen
Sun Microsystems Laboratories
June 11, 2006

Introduction

Language Overview

Basics of Parallelism

Components and APIs

Defining Mathematical Operators

Polymorphism and Type Inference

Parallelism: Generators and Reducers

Contracts, Properties, and Testing

Summary

Context

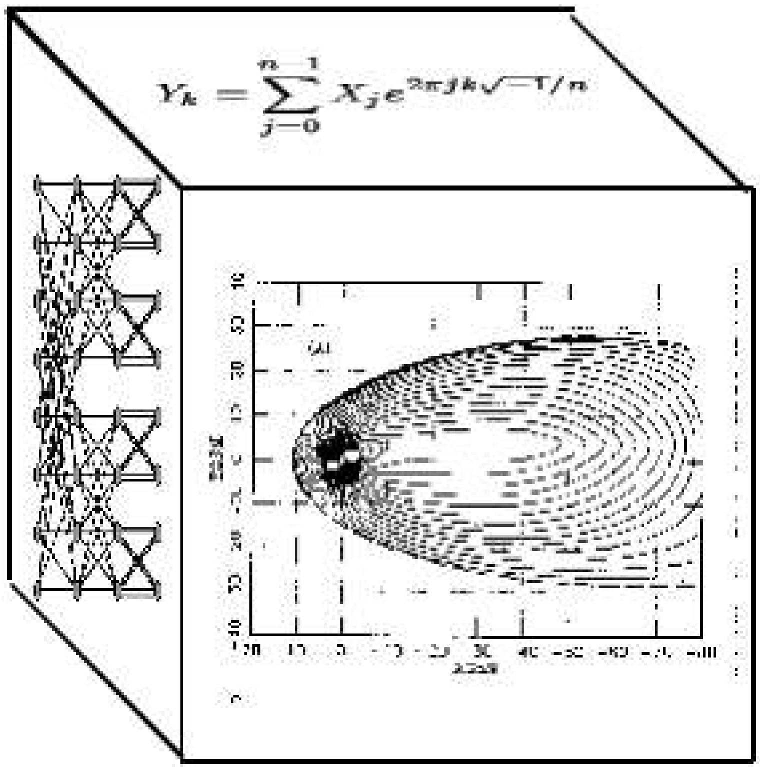
- Improving programmer productivity for scientific and engineering applications
- Research funded in part by the DARPA IPTO (Defense Advanced Research Projects Agency Information Processing Technology Office) through their High Productivity Computing Systems program
- Goal is economically viable technologies for both government and industrial applications by the year 2010 and beyond

Fortress: "To Do for Fortran What Java™ Did for C"

Great ideas from the Java™ programming language:

- Catch "stupid mistakes"
  - > Array bounds and null pointer checking
  - > Automatic storage management
- Platform independence
- Platform-independent multithreading
- Dynamic compilation
- Make programmers more productive

Goal: Science-Centered Computation



- Program structure should reflect the science
- Not FLOPS
- Not communication structure

Key Ideas

- Don't build the language—grow it
- Make programming notation closer to math
- Make parallelism easy to use

Growing a Language

- Languages have gotten much bigger
- You can’t build one all at once
- Therefore it must grow over time
- What happens if you design it to grow?
- How does the need to grow affect the design?
- Need to grow a user community, too

See Steele, “Growing a Language” keynote talk, OOPSLA 1998; *Higher-Order and Symbolic Computation* **12**, 221–236 (1999)

Interesting Language Design Strategy

Wherever possible,
consider whether a proposed language feature
can be provided by a library
rather than having it wired into the compiler.

Making Abstraction Efficient

- We assume implementation technology that makes aggressive use of runtime performance measurement and optimization.
- Repeat the success of the Java™ Virtual Machine
- Goal: programmers (especially library writers) need not fear subroutines, functions, methods, and interfaces for performance reasons
- This may take years, but we're talking 2010

Conventional Mathematical Notation

- The language of mathematics is centuries old, concise, convenient, and widely taught
- Programming language notation can become closer to mathematical notation (Unicode helps a lot)
  - `v_norm = v / ‖v‖`
  - `Σ[k=1:n] a[k] x^k`
  - `C = A ∪ B`
  - `y = 3 x sin x cos 2 x log log x`
- Parsing this stuff is an interesting research problem

Introduction

Language Overview

Basics of Parallelism

Components and APIs

Defining Mathematical Operators

Polymorphism and Type Inference

Parallelism: Generators and Reducers

Contracts, Properties, and Testing

Summary

Syntax

- Goal: what you write on your whiteboard works
- Less clutter, better readability
  - > Type inference
  - > Operator overloading, matching mathematical notation
  - > Noisy punctuation, such as semicolons, is often optional (but we don't rely on indentation)
- Three display/input forms
  - > Displayed Unicode—looks like math
  - > Line-oriented Unicode (use [ ] for subscripts, etc.)
  - > “Twiki-like” mode needs only ASCII (for vi and emacs)

Unicode and Twiki Operator Notation

- Popular operators: `+ - / = < > | { }`
- Abbreviated:
  `[\ \] =/= >= -> => |-> <| |>`
  $[\![\quad]\!] \quad \neq \quad \geq \quad \rightarrow \quad \Rightarrow \quad \mapsto \quad \langle \quad \rangle$
- Short names in all caps:
  `OPLUS DOT TIMES SQCAP AND OR IN`
  $\oplus \quad \cdot \quad \times \quad \sqcap \quad \wedge \quad \vee \quad \in$
- Named: `NORMAL_SUBGROUP_OF` $\lhd$
  (Any full Unicode name may be used.)

Unicode and Twiki Identifier Notation

- Regular: `a zip trickOrTreat foobar`
  a zip $trickOrTreat$ $foobar$
- Formatted:
  `a3 _a a_ a_vec _a_hat a_max foo_bar`
  a_3 $\mathbf{a}$ a $\vec{a}$ $\hat{\mathbf{a}}$ a_{max} $\overline{foo}$
- Greek letters: `alpha beta GAMMA DELTA`
  α β Γ Δ
- Unicode names: `HEBREW_ALEF` א
- Blackboard font:
  `RR QQ NN ZZ ZZ64 RR_star`
  $\mathbb{R}$ $\mathbb{Q}$ $\mathbb{N}$ $\mathbb{Z}$ $\mathbb{Z}64$ $\mathbb{R}^*$

Unicode and Twiki Literal Notation

- Boolean: `true false`
- String: `"Hello, world!"`
  “Hello, world!”
- Numbers: `1234 ffff0000_16`
  1234 $\mathrm{ffff0000}_{16}$
  `6.02 TIMES 10^23`
  6.02×10^{23}
  12345678901234567890645236436352

Numeric Data Types

- Integers ℤ: `23 0 -15245362916252 1266` (signed "big integers" of any size)
- Naturals ℕ: `23 17 1524536216252 1266` (unsigned "big integers" of any size)
- Rational ℚ: `13 5/7 -999/1001`
- Real ℝ, complex ℂ (these include ℤ and ℕ and ℚ)
- Fixed-size integers: ℤ8 ℤ16 ℤ32 ℤ64 ℤ128 ℤ256 ℤ512 ... and ℕ8 ℕ16 ℕ32 ...
- Floating-point: ℝ32 ℝ64 ℝ128 ℝ256 ℝ512 ... and ℂ64 ℂ128 ℂ256 ...

Units and Dimensions

- Units: `m_ kg_ s_ micro_s_ MW_ ns_`
  m kg s $\mu\mathrm{s}$ MW ns
- Dimensions: `Length Mass Time Force`

```
m: RR64 Mass = 3 kg_
_v: RR64[3] Velocity
_p: RR64[3] Momentum
_p := m _v
(* Project v onto p *)

_v := _v (_v DOT _p)/(_p DOT _p)

_v := _p (_v DOT _p)/(_p DOT _p)

_v := _p (_v DOT _p)/(_v DOT _v)
```

$m : \mathbb{R}64\ \mathrm{Mass} = 3\,\mathrm{kg}$

$\mathbf{v} : \mathbb{R}64[3]\ \mathrm{Velocity}$

$\mathbf{p} : \mathbb{R}64[3]\ \mathrm{Momentum}$

$\mathbf{p} := m\,\mathbf{v}$

(* Project v onto p *)

$$\mathbf{v} := \mathbf{v}\frac{\mathbf{v}\cdot\mathbf{p}}{\mathbf{p}\cdot\mathbf{p}}$$

$$\mathbf{v} := \mathbf{p}\frac{\mathbf{v}\cdot\mathbf{p}}{\mathbf{p}\cdot\mathbf{p}}$$

$$\mathbf{v} := \mathbf{p}\frac{\mathbf{v}\cdot\mathbf{p}}{\mathbf{v}\cdot\mathbf{v}}$$

Expressions and Statements

- Everything is an expression
- () is the void value
- Statements are void-typed expressions: **`while`**, **`for`**, assignment, and binding
- Some "statements" may have non-() values: **`if`**, **`do`**, **`atomic`**, **`try`**, **`case`**, **`typecase`**, **`dispatch`**, **`spawn`**

Examples of “Statements”

- `if x ≥ 0 then x else -x end`
- `for k←1#10 do a[k] := k! end`
- `while n<10 do print n; n+=1 end`
- ```
  try
    file = open(fileName)
    process(read(file))
  catch e
    IOException ⇒ handleError(e)
  finally
    close(file)
  end
  ```
  ```

More Examples of “Statements”

- ```
  atomic x := max(x, y[k])
  ```
- ```
  atomic do
     (x, y) := (y, x)
     n_swaps += 1
  end
  ```
- ```
  ans = case n of
            1 ⇒ “unit”
            {2,3,5,7} ⇒ “prime”
            {4,6,8,9} ⇒ “composite”
            else ⇒ “I dunno”
        end
  ```
  ```

Aggregate Expressions

- Set, array, map, and list constants
  ```
  { 2, 3, 5, 7 }
  [ "cat" ↦ "dog", "mouse" ↦ "cat" ]
  ⟨ 0, 1, 1, 2, 3, 5, 8, 13 ⟩
  ```
- Set, array, map, and list comprehensions
  ```
  { x² | x ← primes }
  [ x² ↦ x³ | x ← fibs, x < 1000 ]
  ⟨ x(x+1)/2 | x ← 1#100 ⟩
  ```
- Array pasting
  ```
  [ 1 0
    0 A ]
  ```
  $$\begin{bmatrix} 1 & 0 \\ 0 & A \end{bmatrix}$$

Summation and Other Reductions

- Summation:
```
SUM[k←1:n] a[k] x^k
  Σ[k←1:n] a[k] x^k
```
$$\sum_{k=1}^{n} a_k x^k$$

- Others:
```
∪[k←1:n] S[k]
∩[k←1:n,odd k] S[k]
∧[j←1:m,k←1:n] b[j,k]
∨[k←1:n] b[k]
MAX[k←1:n] a[k]
MIN[k←1:n] a[k]
WEIRDOP[k←1:n] w[k]
```

Binding, Assignment, Generation

- Binding: `v = e`
  Must be a non-final statement within a block
- Assignment: `v := e`
- Generation: `v ← e`
  Used in loops, comprehensions, reductions
- The form `v = e` is also used for equality tests and for keyword arguments; context matters

```
if n = 3 then
   p = pixel(red=ff₁₆,green=33₁₆,blue=cc₁₆)
   drawPixel(p, (x=y), x = 27, y = 19)
end
```

Limited Whitespace Sensitivity

- Subscripting: `a[m n]`
- Scalar times vector: `a [m n]`
- Fractions: `v = 1 / 2 cos x`
  `s = 1/2 g t`^{2}
- Vertical bars: `{ |x| | x←1:20 }`
- Conflicting cues are forbidden:

```
a+b / c+d            (* error *)
a + b / c + d        (* okay *)
a + b/c + d          (* best *)
```

Type System: Objects and Traits

- Traits: like interfaces, but may contain code
  - Based on work by Schärli, Ducasse, Nierstrasz, Black, et al.
- Multiple inheritance of code (but not fields)
  - Objects with fields are the leaves of the hierarchy
- Multiple inheritance of contracts and tests
  - Automated unit testing
- Traits and methods may be parameterized
  - Parameters may be types or compile-time constants
- Primitive types are first-class
  - Booleans, integers, floats, characters are all objects

```
trait Boolean
    extends BooleanAlgebra⟦Boolean,∧,∨,¬,⊻,false,true⟧
    comprises { true, false }
  opr ∧(self, other: Boolean): Boolean
  opr ∨(self, other: Boolean): Boolean
  opr ¬(self): Boolean
  opr ⊻(self, other: Boolean): Boolean
end

object true extends Boolean
  opr ∧(self, other: Boolean) = other
  opr ∨(self, other: Boolean) = self
  opr ¬(self) = false
  opr ⊻(self, other: Boolean) = ¬other
end

object false extends Boolean
  opr ∧(self, other: Boolean) = self
  opr ∨(self, other: Boolean) = other
  opr ¬(self) = true
  opr ⊻(self, other: Boolean) = other
end
```

Parametric Objects

```
object Cart(re: ℝ, im: ℝ) extends ℂ
  opr +(self, other: Cart): Cart =
    Cart(self.re + other.re, self.im + other.im)
  opr -(self): Cart =
    Cart(-self.re, -self.im)
  opr -(self, other: Cart): Cart =
    Cart(self.re - other.re, self.im - other.im)
  opr ·(self, other: Cart): Cart =
    Cart(self.re · other.re - self.im · other.im,
         self.re · other.im + self.im · other.re)
  opr |self| : ℝ = √((self.re)² + (self.im)²)
  ...
end
```

Methods and Fields

- Methods are defined within traits or objects; fields in objects

```
object BankAccount(var balance:ℕ)
  deposit(amount:ℕ) = do
    self.balance += amount
    generateReceipt(amount, balance)
  end
end

myAccount: BankAccount = BankAccount(43)
myReceipt = myAccount.deposit(19)
print myAccount.balance
```

Functions

- Functions are defined at top level or within blocks

```
triple(x:ℝ):ℝ = 3 x

bogglify(n:ℝ):ℝ =
  if n > 3 then
    boggle(x:ℝ) = triple(x+1)
    boggle(47 n + 1) − boggle(n)
  else
    triple n
  end
```

Simple Example: NAS CG Kernel (ASCII)

```
conjGrad(A: Matrix[\Float\], x: Vector[\Float\]):
        (Vector[\Float\], Float) = do
  cgit_max = 25
  z: Vector[\Float\] = 0
  r: Vector[\Float\] = x
  p: Vector[\Float\] = r
  rho: Float = r^T r
  for j <- seq(1:cgit_max) do
    q = A p
    alpha = rho / p^T q
    z := z + alpha p
    r := r - alpha q
    rho0 = rho
    rho := r^T r
    beta = rho / rho0
    p := r + beta p
  end
  (z, ||x - A z||)
end

(z,norm) = conjGrad(A,x)
```

Matrix[\T\] and Vector[\T\] are parameterized interfaces, where T is the type of the elements.

The form x:T=e declares a variable x of type T with initial value e, and that variable may be updated using the assignment operator :=.

Simple Example: NAS CG Kernel (ASCII)

```
conjGrad[\Elt extends Number, nat N,
          Mat extends Matrix[\Elt,N BY N\],
          Vec extends Vector[\Elt,N\]
        \](A: Mat, x: Vec): (Vec, Elt) = do
  cgit_max = 25
  z: Vec = 0
  r: Vec = x
  p: Vec = r
  rho: Elt = r^T r
  for j <- seq(1:cgit_max) do
    q = A p
    alpha = rho / p^T q
    z := z + alpha p
    r := r - alpha q
    rho0 = rho
    rho := r^T r
    beta = rho / rho0
    p := r + beta p
  end
  (z, ||x - A z||)
end

(z,norm) = conjGrad(A,x)
```

Here we make conjGrad a generic procedure. The runtime compiler may produce multiple instantiations of the code for various types Elt.

The form x=e as a statement declares variable x to have an unchanging value. The type of x is exactly the type of the expression e.

Simple Example: NAS CG Kernel (Unicode)

```
conjGrad⟦Elt extends Number, nat N,
         Mat extends Matrix⟦Elt,N×N⟧,
         Vec extends Vector⟦Elt,N⟧
        ⟧(A: Mat, x: Vec): (Vec, Elt) = do
  cgit_max = 25
  z: Vec = 0
  r: Vec = x
  p: Vec = r
  ρ: Elt = r^T r
  for j ← seq(1:cgit_max) do
     q = A p
     α = ρ / p^T q
     z := z + α p
     r := r - α q
     ρ0 = ρ
     ρ := r^T r
     β = ρ / ρ0
     p := r + β p
  end
  (z, ‖x - A z‖)
end
```

This would be considered entirely equivalent to the previous version. You might think of this as an abbreviated form of the ASCII version, or you might think of the ASCII version as a way to conveniently enter this version on a standard keyboard.

Simple Example: NAS CG Kernel

$$
\begin{array}{l}
\mathit{conjGrad}[\![\mathrm{Elt}\ \mathbf{extends}\ \mathrm{Number},\ \mathbf{nat}\ \mathrm{N}, \\
\qquad\qquad \mathrm{Mat}\ \mathbf{extends}\ \mathrm{Matrix}[\![\mathrm{Elt}, N \times N]\!], \\
\qquad\qquad \mathrm{Vec}\ \mathbf{extends}\ \mathrm{Vector}[\![\mathrm{Elt}, \mathrm{N}]\!] \\
\qquad\quad]\!](A : \mathrm{Mat},\ x : \mathrm{Vec}) : (\mathrm{Vec},\ \mathrm{Elt}) \\
\quad \mathit{cgit}_{\max} = 25 \\
\quad z : \mathrm{Vec} = 0 \\
\quad r : \mathrm{Vec} = x \\
\quad p : \mathrm{Vec} = r \\
\quad \rho : \mathrm{Elt} = r^{\mathrm{T}} r \\
\quad \mathbf{for}\ j \leftarrow \mathbf{seq}(1 : \mathit{cgit}_{\max})\ \mathbf{do} \\
\qquad q = A\,p \\
\qquad \alpha = \dfrac{\rho}{p^{\mathrm{T}} q} \\
\qquad z := z + \alpha\,p \\
\qquad r := r - \alpha\,q \\
\qquad \rho_0 = \rho \\
\qquad \rho := r^{\mathrm{T}} r \\
\qquad \beta = \dfrac{\rho}{\rho_0} \\
\qquad p := r + \beta\,p \\
\quad \mathbf{end} \\
\quad (z,\ \|x - A\,z\|)
\end{array}
$$

It's not new or surprising that code written in a programming language might be displayed in a conventional math-like format. The point of this example is how similar the code is to the math notation: the gap between the two syntaxes is relatively small. We want to see what will happen if a principal goal of a new language design is to minimize this gap.

Comparison: NAS NPB 1 Specification

$$
\begin{aligned}
& z = 0 \\
& r = x \\
& \rho = r^T r \\
& p = r \\
& \textbf{DO } i = 1, 25 \\
& \qquad q = A\,p \\
& \qquad \alpha = \rho / (p^T q) \\
& \qquad z = z + \alpha\,p \\
& \qquad \rho_0 = \rho \\
& \qquad r = r - \alpha\,q \\
& \qquad \rho = r^T r \\
& \qquad \beta = \rho / \rho_0 \\
& \qquad p = r + \beta\,p \\
& \textbf{ENDDO} \\
& \text{compute residual norm explicitly: } \|r\| = \|x - A\,z\|
\end{aligned}
$$

$$
\begin{aligned}
& z : \mathrm{Vec} = 0 \\
& r : \mathrm{Vec} = x \\
& p : \mathrm{Vec} = r \\
& \rho : \mathrm{Elt} = r^{\mathrm{T}} r \\
& \textbf{for } j \leftarrow \textbf{seq}(1 : cgit_{\max})\ \textbf{do} \\
& \quad q = A\,p \\
& \quad \alpha = \frac{\rho}{p^{\mathrm{T}} q} \\
& \quad z := z + \alpha\,p \\
& \quad r := r - \alpha\,q \\
& \quad \rho_0 = \rho \\
& \quad \rho := r^{\mathrm{T}} r \\
& \quad \beta = \frac{\rho}{\rho_0} \\
& \quad p := r + \beta\,p \\
& \textbf{end} \\
& (z, \|x - A\,z\|)
\end{aligned}
$$

Comparison: NAS NPB 2.3 Serial Code

```
do j=1,naa+1
   q(j) = 0.0d0
   z(j) = 0.0d0
   r(j) = x(j)
   p(j) = r(j)
   w(j) = 0.0d0
enddo
sum = 0.0d0
do j=1,lastcol-firstcol+1
   sum = sum + r(j)*r(j)
enddo
rho = sum
do cgit = 1,cgitmax
   do j=1,lastrow-firstrow+1
      sum = 0.d0
      do k=rowstr(j),rowstr(j+1)-1
         sum = sum + a(k)*p(colidx(k))
      enddo
      w(j) = sum
   enddo
   do j=1,lastcol-firstcol+1
      q(j) = w(j)
   enddo
   do j=1,lastcol-firstcol+1
      w(j) = 0.0d0
   enddo
   sum = 0.0d0
   do j=1,lastcol-firstcol+1
      sum = sum + p(j)*q(j)
   enddo
   d = sum
   alpha = rho / d
   rho0 = rho
   do j=1,lastcol-firstcol+1
      z(j) = z(j) + alpha*p(j)
      r(j) = r(j) - alpha*q(j)
   enddo
   sum = 0.0d0
   do j=1,lastcol-firstcol+1
      sum = sum + r(j)*r(j)
   enddo
   rho = sum
   beta = rho / rho0
   do j=1,lastcol-firstcol+1
      p(j) = r(j) + beta*p(j)
   enddo
enddo
do j=1,lastrow-firstrow+1
   sum = 0.d0
   do k=rowstr(j),rowstr(j+1)-1
      sum = sum + a(k)*z(colidx(k))
   enddo
   w(j) = sum
enddo
do j=1,lastcol-firstcol+1
   r(j) = w(j)
enddo
sum = 0.0d0
do j=1,lastcol-firstcol+1
   d   = x(j) - r(j)
   sum = sum + d*d
enddo
d = sum
rnorm = sqrt( d )
```

Introduction

Language Overview

Basics of Parallelism

Components and APIs

Defining Mathematical Operators

Polymorphism and Type Inference

Parallelism: Generators and Reducers

Contracts, Properties, and Testing

Summary

Parallelism Is Not a Feature!

- Parallel programming is not a goal, but a pragmatic compromise.
- It would be a lot easier to program a single processor chip running at 1 PHz than a million processors running at 20 GHz.
  - We don't know how to build a 1 Phz processor.
  - Even if we did, someone would still want to strap a bunch of them together!
- Parallel programming is difficult and error-prone. (This is not a property of machines, but of people.)

Should Parallelism Be the Default?

- “Loop” can be a misleading term
  - > A set of executions of a parameterized block of code
  - > Whether to order or parallelize those executions should be a separate question
  - > Maybe you should have to ask for sequential execution!
- Fortress “loops” are parallel by default
  - > This is actually a library convention about generators

In Fortress, Parallelism Is the Default

```
for i←1:m, j←1:n do
  a[i,j] := b[i] c[j]
end
```

`1:n` is a generator

```
for i←seq(1:m) do
  for j←seq(1:n) do
    print a[i,j]
  end
end
```

`seq(1:m)` is a sequential generator

```
for (i,j)←a.indices do
  a[i,j] := b[i] c[j]
end
```

`a.indices` is a generator for the indices of the array **`a`**

`a.indices.rowMajor` is a sequential generator of indices

```
for (i,j)←a.indices.rowMajor do
  print a[i,j]
end
```

Primitive Constructs for Parallelism

- Iterations of a **for** loop

```
for x ← 1#1000 do
  a[x] := x
end
```

- Tuples

```
(a1, a2, a3) = (e1, e2, e3)
f(e1, e2)
```

- Spawned threads

```
t1 = spawn do e1 end
t2 = spawn do e2 end
a1 = t1.value()
a2 = t2.value()
```

Give rise to implicit threads

Give rise to explicit spawned threads

Array Types

- May include bounds, or leave them optional

```
a : RR64[xSize, ySize, zSize]
M : Complex⟦RR64⟧[ 32, 32 ]
space : Vector⟦RR64,6⟧[:,:,:]
```

- Bounds are specified using **nat** type parameters:

```
conjGrad⟦Elt extends Number, nat N,
         Mat extends Matrix⟦Elt,N×N⟧,
         Vec extends Vector⟦Elt,N⟧
         ⟧(A: Mat, x: Vec): (Vec, Elt)
```

- Both **Matrix** and **Vector** implement **Array**

Constructing Arrays

- Construct using an aggregate constant:

```
identity = [ 1 0
             0 1 ]
```

- Or a comprehension:

```
a = [(x, y, 1) ↦ 0.0 | x ← 1 : xSize ,
                       y ← 1 : ySize
     (1, y, z) ↦ 0.0 | y ← 1 : ySize ,
                       z ← 2 : zSize
     (x, 1, z) ↦ 0.0 | x ← 2 : xSize ,
                       z ← 2 : zSize
     (x, y, z) ↦ x + y · z | x ← 2 : xSize ,
                             y ← 2 : ySize ,
                             z ← 2 : zSize ]
```

Indexing and Assignment

- Specified by the trait **`Indexable`**:

```
trait Indexable⟦Self extends Indexable⟦Self,E,I⟧,
                E extends Object, I extends Object⟧
extends Object
  opr [ i : I ] : E
  opr [ i : I ]:=( e : E ) : ()
end

trait Array⟦E extends Object, I extends ArrayIndex⟧
extends Indexable⟦ Array⟦E, I⟧, E, I ⟧
  ...
end
```

- The type notation **`T[a,b]`** is simply shorthand for **`Array⟦T,(a,b)⟧`**

Generators

- Generators (defined by libraries) manage parallelism and the assignment of threads to processors
- Aggregates
  - Lists ⟨`1,2,4,3,4`⟩ and vectors `[1 2 4 3 4]`
  - Sets `{1,2,3,4}` and multisets `{|1,2,3,4,4|}`
  - Arrays (including multidimensional)
- Ranges `1:10` and `1:99:2` and `0#50`
- Index sets `a.indices` and `a.indices.rowMajor`
- Index-value sets `ht.keyValuePairs`

Local Variables, Reduction Variables

- Variables unassigned in a loop body are *local*
- Variables accumulated in a loop body but not read are *reduction variables*

```
meanVar⟦E extends Number, I extends ArrayIndex⟧
       (a : E[I]): (E,E) = do
   n     : E := 0
   sum   : E := 0
   sumsq : E := 0
   for i ← a.indices do
     n += 1
     t = a[i]
     sum   += t
     sumsq += t t
   end
   (sum/n, (sumsq − sum sum)/n )
end
```

Atomic Blocks

- Variables mutated in a loop body, but not reduced, must be accessed within an atomic block.

```
histogram⟦nat lo, nat sz⟧
         (a: A[#,#]): Int[lo#sz] =
do hist : Int[lo#sz] := 0
   for i,j ← a.indices do
      atomic do
        hist[a[i,j]] += 1
      end
   end
   hist
end
```

Introduction

Language Overview

Basics of Parallelism

Components and APIs

Defining Mathematical Operators

Polymorphism and Type Inference

Parallelism: Generators and Reducers

Contracts, Properties, and Testing

Summary

Replaceable Components

- Avoid a monolithic “Standard Library”
- Replaceable components with version control
- Encourage alternate implementations
  - Performance choices
  - Test them against each other
- Encourage experimentation
  - Framework for alternate language designs

Encapsulated Upgradable Components

- The stability of static linking
- The sharing and upgradability of dynamic linking

Desired Properties

- Installation never blocked by existing components
- Execution without signaling a component error
- Upgrade without affecting other applications
- No unnecessary copies

Hello World

Types checked against API

```
component Hello
   import print from IO
   export Executable
   run(args: String...) = print “Hello world”
end
```

`Hello` imports this

```
api IO
  print: String → ()
end
```

`Hello` exports this

```
api Executable
  run(args: String...) → ()
end
```

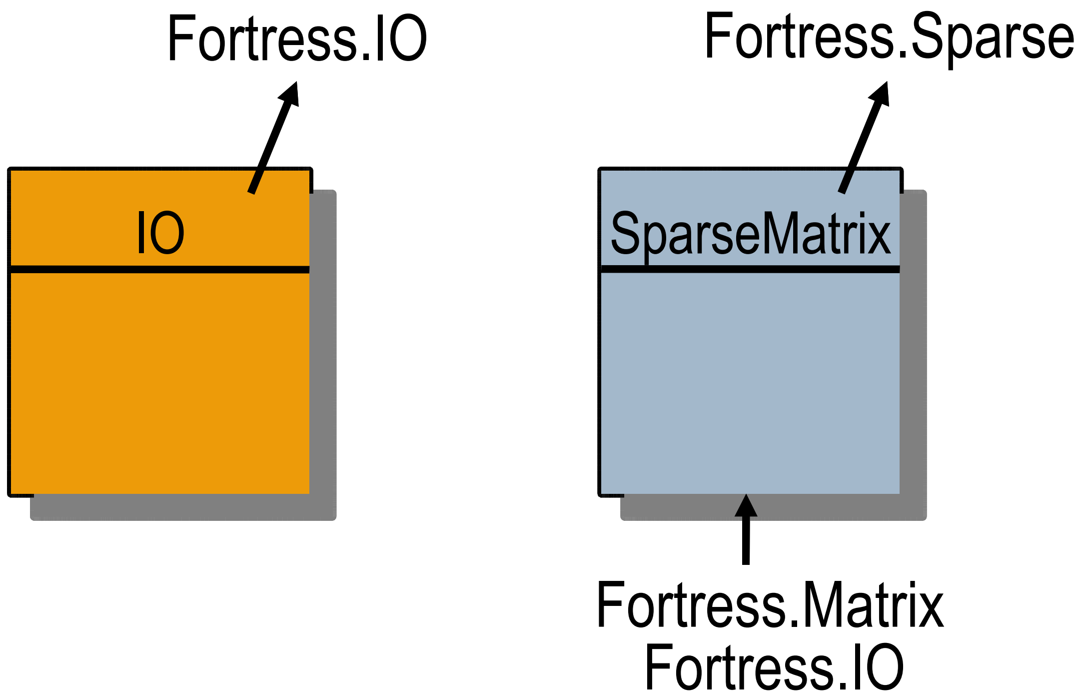
APIs

- **APIs** are the “interfaces” of components.
- APIs consist only of **declarations**, not definitions.
- An API **imports** other APIs it uses.
- Each API in the world has a distinct name.

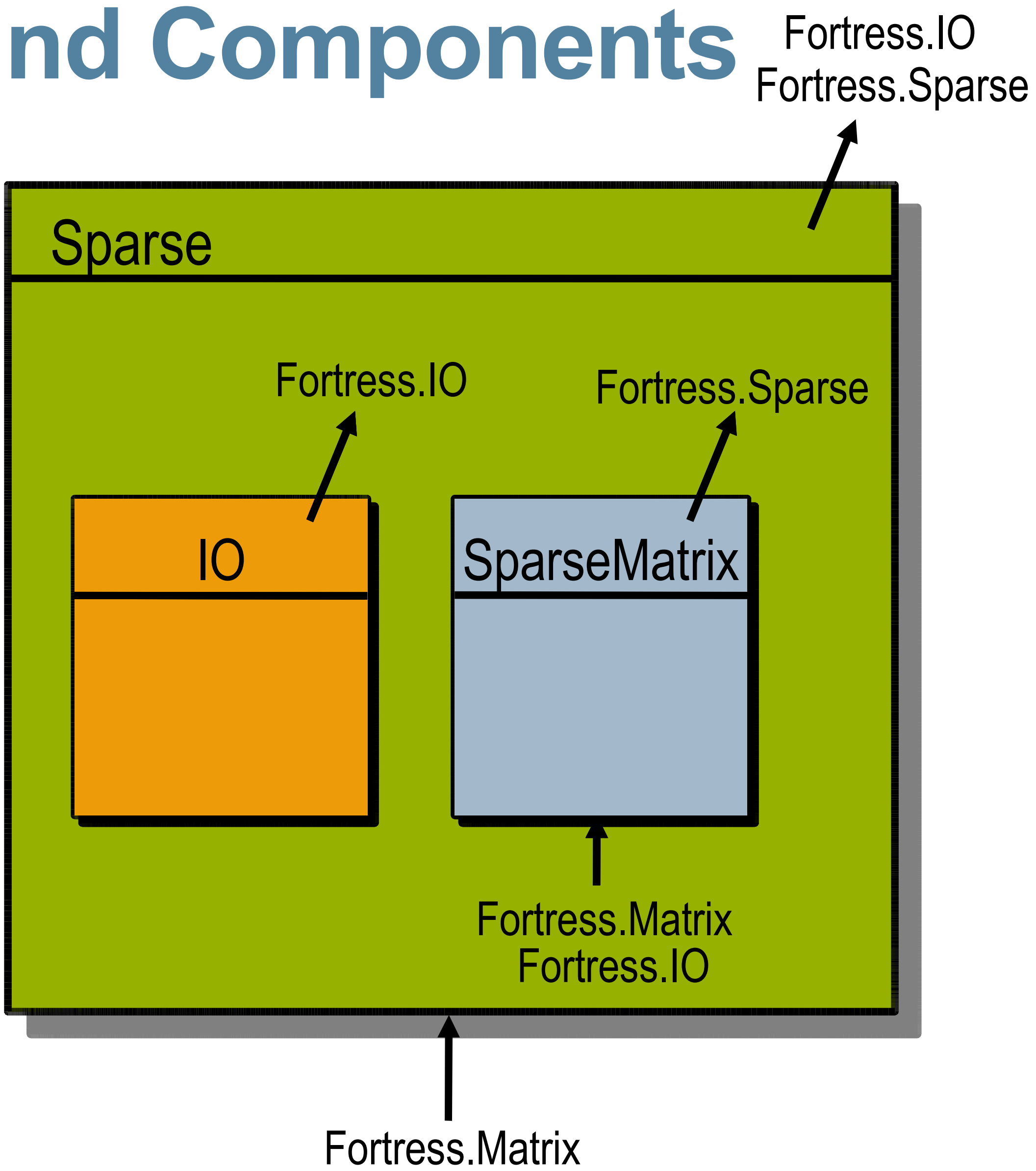
Components

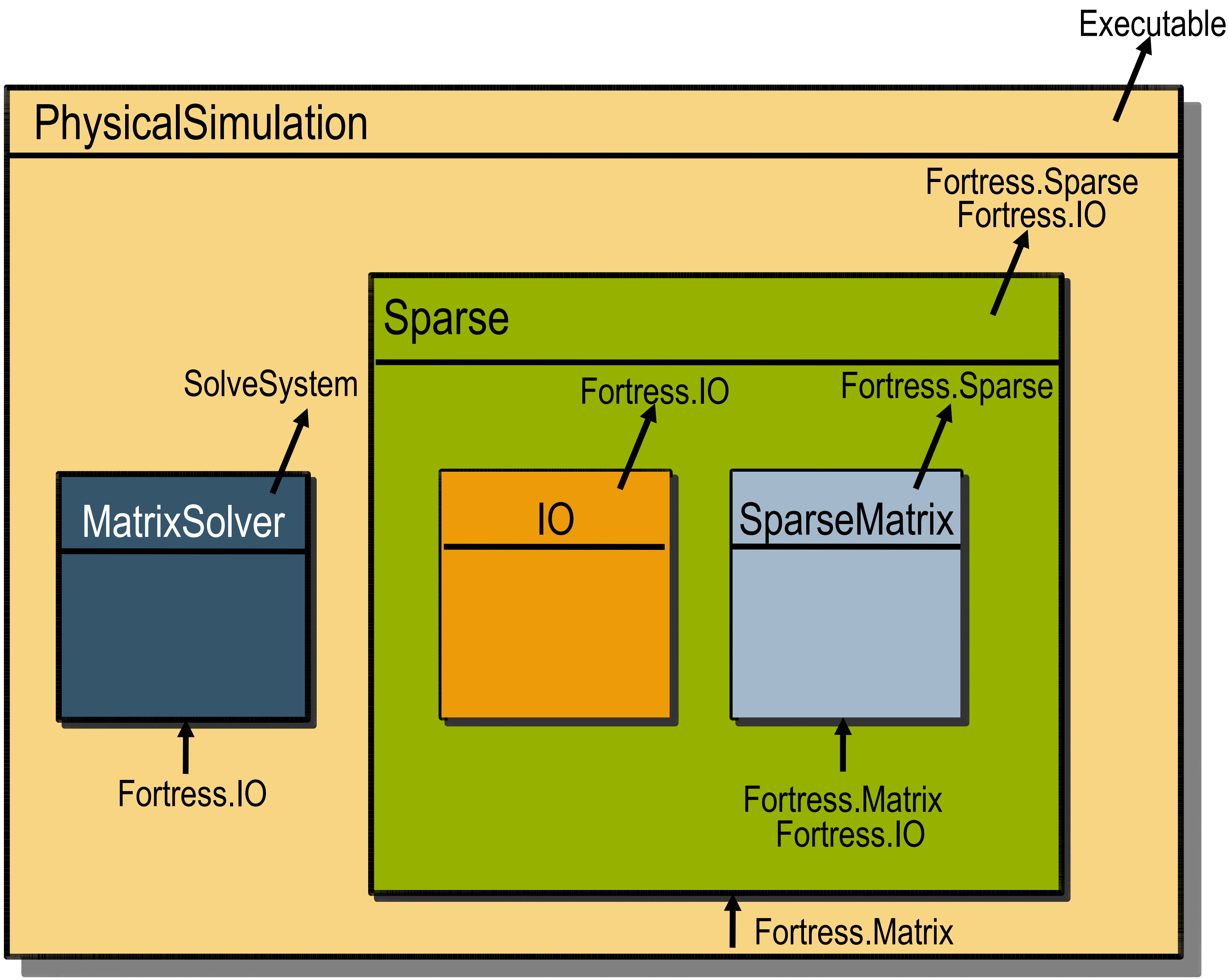
- Components are immutable
- **Simple components** are units of compilation
  - > Typically the size of small Java packages
- **Compound components** are produced by combining components
  - > Through linking
  - > Through upgrade
- Components import and export **APIs**

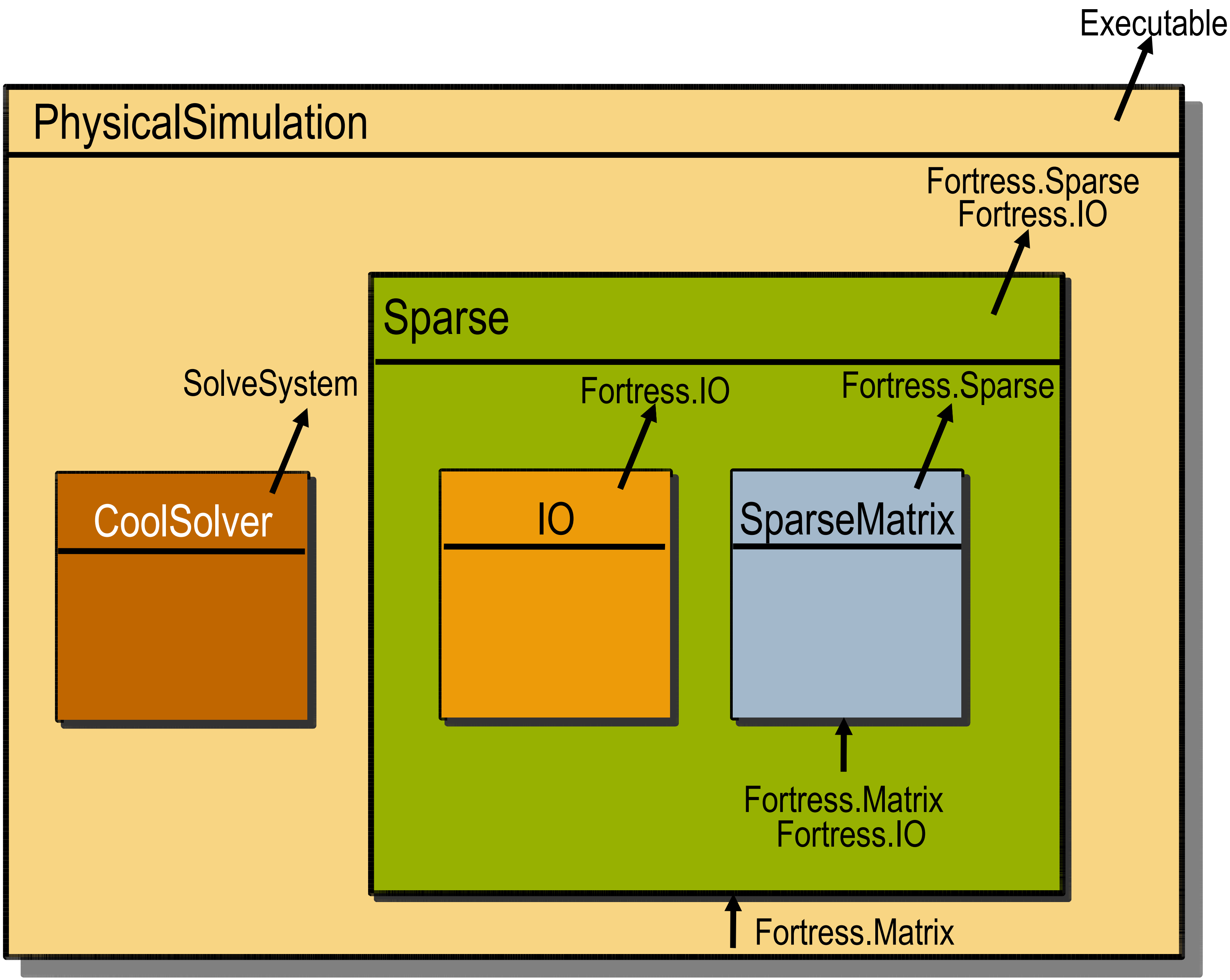
Simple Components



Compound Components




Executable
PhysicalSimulation
Fortress.Sparse
Fortress.IO
Sparse
SolveSystem
Fortress.IO
Fortress.Sparse
MatrixSolver
IO
SparseMatrix
Fortress.IO
Fortress.Matrix
Fortress.IO
Fortress.Matrix


Executable
PhysicalSimulation
Fortress.Sparse
Fortress.IO
Sparse
SolveSystem
Fortress.IO
Fortress.Sparse
CoolSolver
IO
SparseMatrix
Fortress.Matrix
Fortress.IO
Fortress.Matrix

Sharing: Fortresses

- Components are not manipulated directly; they are stored in **fortresses**.
- Fortresses are persistent databases mapping names to components and APIs.
- Typically, a single machine includes a single fortress.

Efficient Implementation

- Because components are immutable, they can be shared at will.
- Components not directly reachable can be referred to by other compound components.
- Reclamation of unused components can be handled via conventional garbage collection.

Introduction
Language Overview
Basics of Parallelism
Components and APIs
Defining Mathematical Operators
Polymorphism and Type Inference
Parallelism: Generators and Reducers
Contracts, Properties, and Testing
Summary

What Syntax Is Actually Wired in?

- Parentheses ( ) for grouping
- Comma , to separate expressions in tuples
- Semicolon ; to separate statements on a line
- Dot . for field and method selection
- Juxtaposition is a binary operator
- Any other operator can be infix, prefix, and/or postfix
- Many sets of brackets
- Conservative, traditional rules of precedence
  - A dag, not always transitive (examples: **A+B>C** is okay; so is **B>C∨D>E**; but **A+B∨C** needs parentheses)

Libraries Define . . .

- Which operators have infix, prefix, postfix definitions, and what types they apply to:

```
opr -(m:ℤ64,n:ℤ64) = m.subtract(n)
opr -(m:ℤ64) = m.negate()
opr (n:ℤ64)! = if n=0 then 1 else n·(n-1)! end
```

- Whether a juxtaposition is meaningful:

```
opr juxtaposition(m:ℤ64,n:ℤ64) = m.times(n)
```

- What bracketing operators actually mean:

```
opr ⌈x:Number⌉ = ceiling(x)
opr |x:Number| = if x<0 then -x else x end
opr |s:Set⟦T⟧| = s.size
```

But Wasn't Operator Overloading a Disaster in C++ ?

- Yes, it was
  - > Not enough operators to go around
  - > Failure to stick to traditional meanings
- We have also been tempted and had to resist
- We see benefits in using notations for programming that are also used for specification

Matrix-Vector Multiplication

- We want to define operators within traits.
  - > Good for providing multiple implementations for a data type
  - > Good for enforcing contracts on subtypes (and therefore enforcing contracts on the multiple implementations)
- We want nice notation, not `x.multiply(y)`
- We want to define both Vector-times-Matrix and Matrix-times-Vector in the Matrix trait.

Functional Methods

- A functional method declaration has an explicit **self** parameter in the parameter list, rather than an implicit **self** parameter before the method name
- A functional method invocation uses the same syntax as function calls
- Example:

```
trait Vector
  opr +(self, other:Vector):Vector
  double(self): Vector = self + self
  ...
end
x = v1 + double(v2)
```

Juxtaposition Operator

- Juxtaposition is an infix operator in Fortress.
  - When the left operand is a function, juxtaposition performs function application.
  - When the left operand is a number, juxtaposition performs multiplication.
  - When the left operand is a string, juxtaposition performs string concatenation.
- Example:

```
y = 3 x sin x cos 2 x log log x
```

Operator Overloading

- Operator overloading allows multiple operator declarations with the same operator name.
- Operator applications are equivalent in behavior to function calls.
- Example:

```
opr  ⦇x:Number         ⦈ : Number = x²
opr  ⦇x:Number,y:Number⦈ : Number = x² + y²

 ⦇3  ⦈      (* reduces to  9 *)
 ⦇3, 4⦈     (* reduces to 25 *)
```

Restrictions on Overloading

No undefined or ambiguous calls at run time

- No statically ambiguous function calls
- No dynamically ambiguous function calls
  - Fortress performs multi-argument dispatch
  - But a special rule forbids even potential ambiguity
- Theorem: If there is a statically most specific applicable declaration, then there is a dynamically most specific applicable declaration.

Overloading and Subtyping

- Assuming that ℤ64 is a subtype of `Number`, the following two declarations are ambiguous:

```
foo(x:Number, y:ℤ64)
foo(x:ℤ64, y:Number)
```

The following new declaration would resolve the ambiguity:

```
foo(x:ℤ64, y:ℤ64)
```

Matrix-Vector Multiplication in Fortress

```
trait Matrix⟦T⟧ excludes { Vector⟦T⟧ }
  ...
  opr juxtaposition(self, other:Vector⟦T⟧)
  opr juxtaposition(other:Vector⟦T⟧, self)
end
```

```
x = v M + M v
```

Introduction
Language Overview
Basics of Parallelism

Components and APIs
Defining Mathematical Operators
Polymorphism and Type Inference

Parallelism: Generators and Reducers
Contracts, Properties, and Testing
Summary

Subtype Polymorphism

- Subtype polymorphism allows code reuse.

```
object Container(var element:Object)
  setElement(e:Object):() = element := e
  getElement():Object = element
end
```

  - Storing: safe upcasts

```
c = Container(0)
c.setElement(2)
```

  - Retrieving: potentially unsafe downcasts

```
x: ℤ64 = cast⟦ℤ64⟧(c.getElement())
```

Parametric Polymorphism

- Parametric polymorphism also allows code reuse.

```
object Container⟦T extends Equality⟧
    (var element:T)
  setElement(e:T):() = element := e
  getElement():T = element
end

c = Container⟦ℤ64⟧(0)
c.setElement(2)

x: ℤ64 = c.getElement()
```

Static Parameters

Parameters may be types or compile-time constants.

- Type parameters
  - types such as traits, tuple types, and arrow types
- **`int`** and **`nat`** parameters
  - integer values (**`nat`** parameters are non-negative)
- **`bool`** parameters
  - Boolean values
- **`dim`** and **`unit`** parameters
  - dimensions and units
- **`opr`** and **`nam`** parameters
  - operator symbols and method names

Parameterized Functions and Traits

- Functions, methods, traits, and objects are allowed to be parametric with respect to static parameters.

```
makeList⟦T, nat length⟧(rest:T[length]) =
   if length = 0 then Empty
   else Cons(rest[0],
               makeList(rest[1#(length-1)]))
   end
```

Nat and Bool Parameters

- Nat parameters

```
f⟦nat n⟧(x:ℝ64 Length^(2n)):
            ℝ64 Length^n = √x
```

- Bool parameters

```
trait RationalQuantity⟦bool negativeInf,
                       bool positiveInf,
                       bool nan⟧
  ...
end
```

Dimension/Unit/Operator/Name Parameters

- Dimension and unit parameters

```
opr √⟦unit U⟧(x: ℝ64 U²):ℝ64 U =
  numericalsqrt(x/U²) U
```

- Operator and name parameters

```
trait CommutativeMonoid⟦T,opr ⊙,nam id⟧
  ...
end
```

The "Self Types" Trick

```
trait Equality⟦T extends Equality⟦T⟧ ⟧
  opr =(self, T):Boolean
end

trait Ordering⟦T extends Ordering⟦T⟧ ⟧
  extends Equality⟦T⟧
    opr ≤(self, other: T):Boolean
    opr ≥(self, other: T) = other ≤ self
    opr <(self, other: T) = not (other ≤ self)
    opr >(self, other: T) = not (self ≤ other)
    opr CMP(self, other: T) =
      if self > other then GreaterThan
      elif self < other then LessThan
      else EqualTo end
end
```

Idiom for self typing

Interesting Uses of Types at Runtime

- Operations dependent on type parameters

```
cast⟦T⟧(x : Object): T =
   typecase x in
     T ⇒ x
     else ⇒ throw CastException
   end
```

Here x : T

Interesting Type Relationships

- Elimination of redundant parameters
- Infinitely broad extensions
  - Monomorphic extension of polymorphic types
- Variant subtyping
  - Covariant subtyping
  - Contravariant subtyping
- Unifying concept: `where` clauses

Elimination of Redundant Parameters

- Instead of:

```
trait Unit⟦D extends Dimension⟧
trait Measurement⟦D extends Dimension,
                  U extends Unit⟦D⟧ ⟧
```

- We can write:

```
trait Unit⟦D extends Dimension⟧
trait Measurement⟦U extends Unit⟦D⟧ ⟧
      where {D extends Dimension}
```

Infinitely Broad Extensions

- We can define a single type Empty that is a subtype of all lists:

```
trait List⟦T⟧

object Cons⟦T⟧(first:T, rest:List⟦T⟧)
       extends List⟦T⟧

object Empty extends List⟦T⟧
       where {T extends Object}
```

Variant Subtyping

- We can define covariant lists without additional language constructs:

```
trait List⟦X extends Y⟧ extends List⟦Y⟧
    where {Y extends Object}
  cons(y:Y):List⟦Y⟧ = Cons⟦Y⟧(y, self)
  ...
end
```

Type inference can fill this type in

```
x : List⟦Number⟧ = Empty.cons(3).cons(5.7)
```

Introduction
Language Overview
Basics of Parallelism

Components and APIs
Defining Mathematical Operators
Polymorphism and Type Inference

Parallelism: Generators and Reducers
Contracts, Properties, and Testing
Summary

Data and Control Models

- Data model: shared global address space
- Control model: multithreaded
  - Basic primitives are tuples and **`spawn`**
  - We hope application code seldom uses **`spawn`**
- Declared distribution of data and threads
  - Managing aggregates integrated into type system
  - Policies programmed as libraries, not wired in
- Transactional access to shared variables
  - Atomic blocks
  - Explicit testing and signaling of failure/retry
  - Deadlock-free, minimize blocking

Data and Control Locality

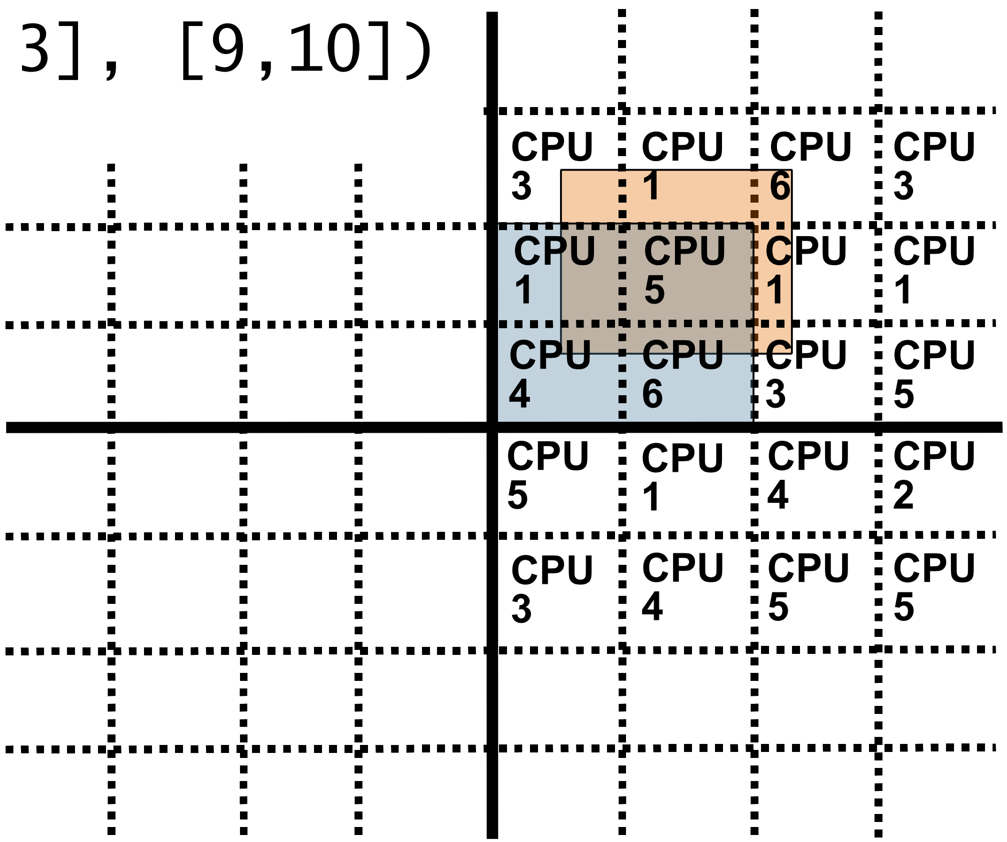
```
for i ← 1#1000 do
  a[i] := a[i] + b[i]
end

for i ← 1#1000 do
  a[i] := a[i] / c[i]
end
```

- Opportunities for locality:
  - Co-locate chunks of arrays a, b, and c
  - Co-locate iterations of the loops (both manipulate the same array a)

Distributions: Allocating Data

Allocation: `d.array(l, u)`

```
a := d.array([0,0], [8,8])
b := d.array([2,3], [9,10])
```



Placing Computations

- We can:
  - Co-locate data by using a common distribution
  - Find the region of an object by using its `region` method
- But how do we place a computation on a specific region of the machine?
  - Augment the spawn expression:

```
spawn x.region do f(x) end
```

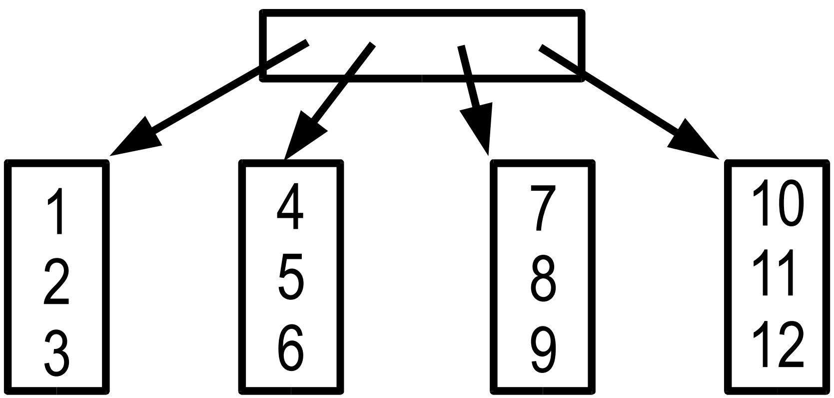
Revisiting Our Example

```
a = d.array([1],[1000])
b = d.array([1],[1000])
c = d.array([1],[1000])

for i ← a.indices() do
  a[i] := a[i] + b[i]
end
for i ← a.indices() do
  a[i] := a[i] / c[i]
end
```

- Opportunities for locality:
  - Co-locate chunks of arrays a, b, and c
  - Co-locate iterations of the loops (both manipulate the same array a)

Distributions

- Describe how to map a data structure onto a region
  - Block, block-cyclic, etc., and user-definable!
  - Map an array into a chip? Use a local heap.
  - Map an array onto a cluster? Break it up.

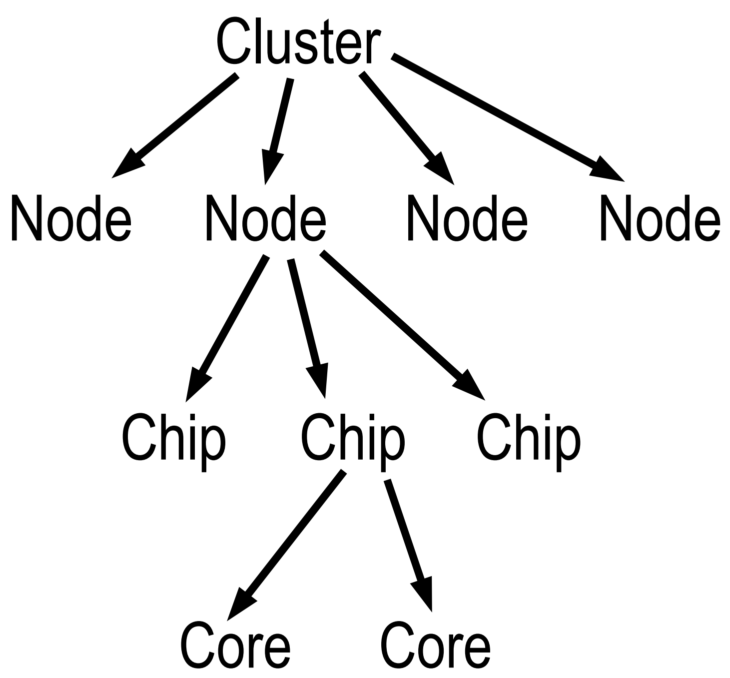


Some Example Distributions

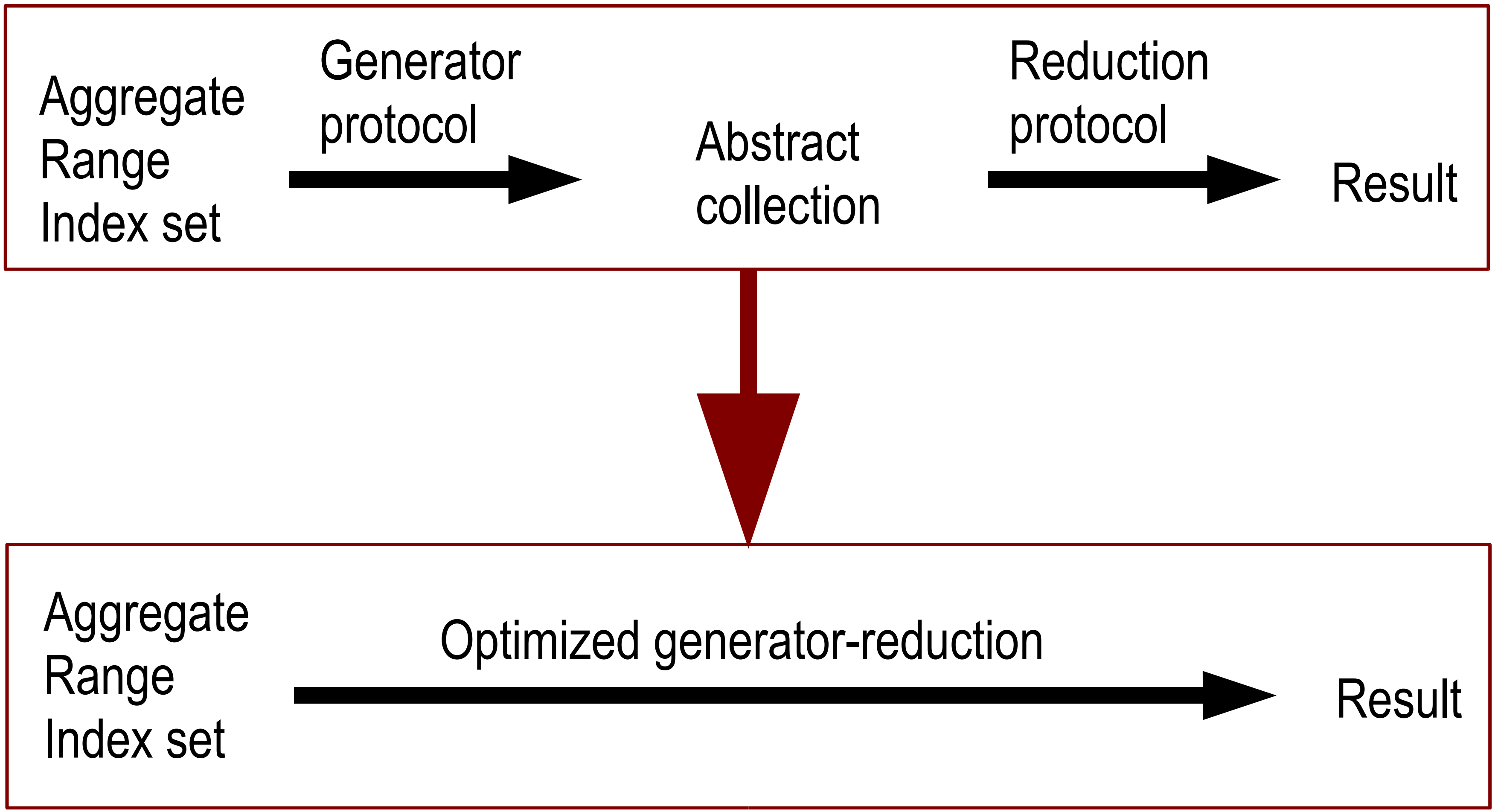
`default`	Used when no other distribution given
`seq(d)`	Data distributed, computation sequential
`local`	Data local, computation sequential
`par`	Chunks of size 1, no particular layout
`ruler`	Hierarchical division at powers of 2
`morton`	Morton order, Z-layout
`blockCyclic(n)`	Block cyclic, block size n
`blocked(n)`	Blocked, block size multiple of n
`rowMajor(d)`	Uninterleave dimensions
`columnMajor(d)`	

Regions

- Hierarchical data structure describes CPU and memory resources and their properties
  - > Allocation heaps
  - > Parallelism
  - > Memory coherence
- A running thread can find out its resources
- Spawn takes an optional region argument
- Distribution assigns regions



Abstract Collections



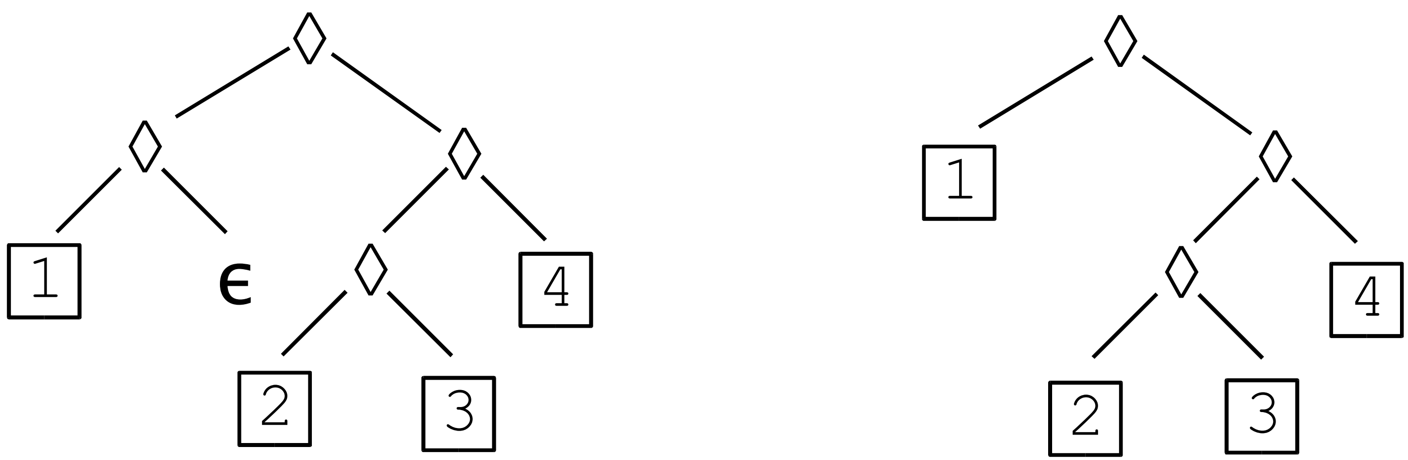
Representation of Abstract Collections

Binary operator ◊

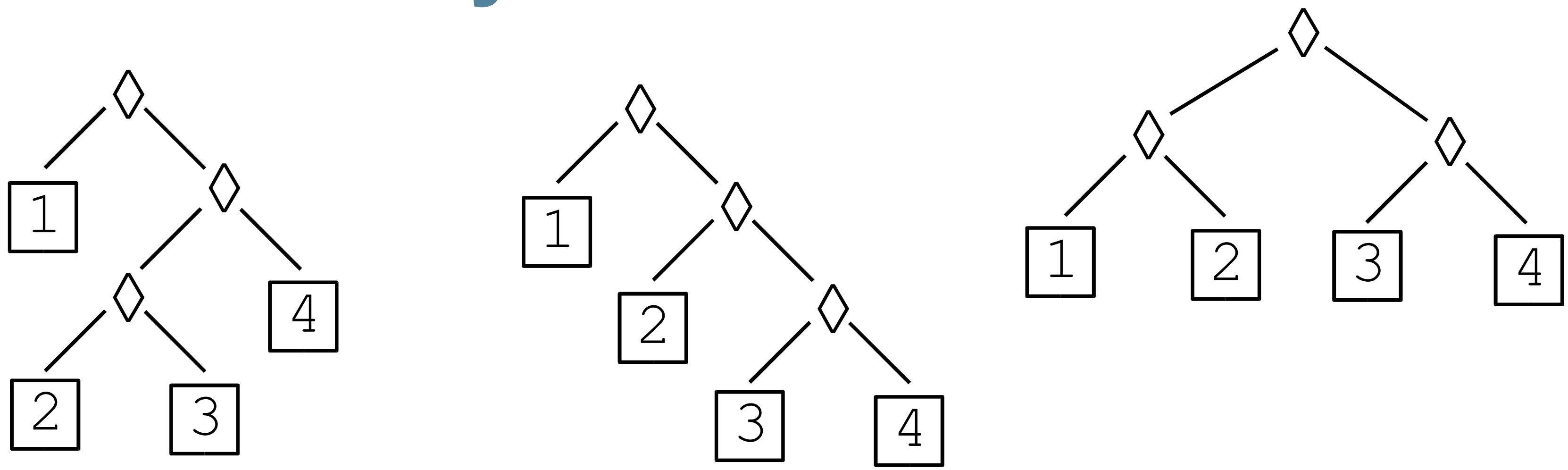
Leaf operator ("unit") □

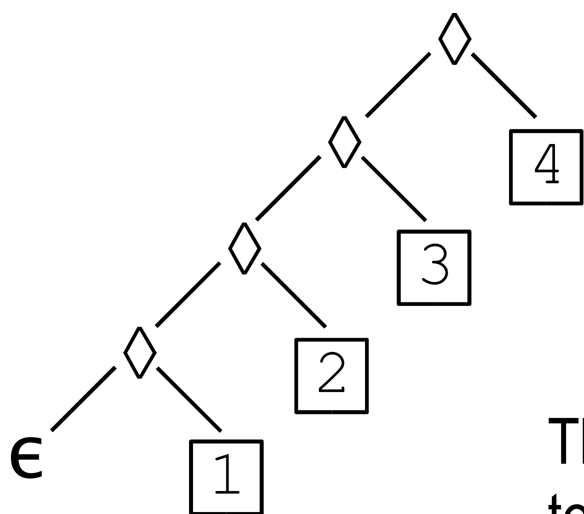
Optional empty collection ("zero") ϵ

that is the identity for ◊

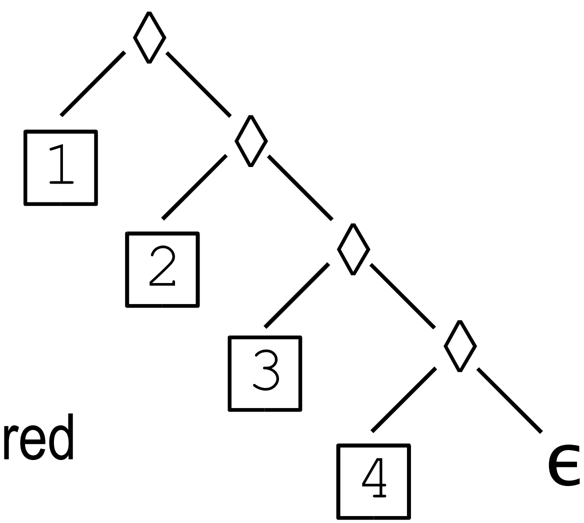


Associativity





These are all considered to be equivalent.



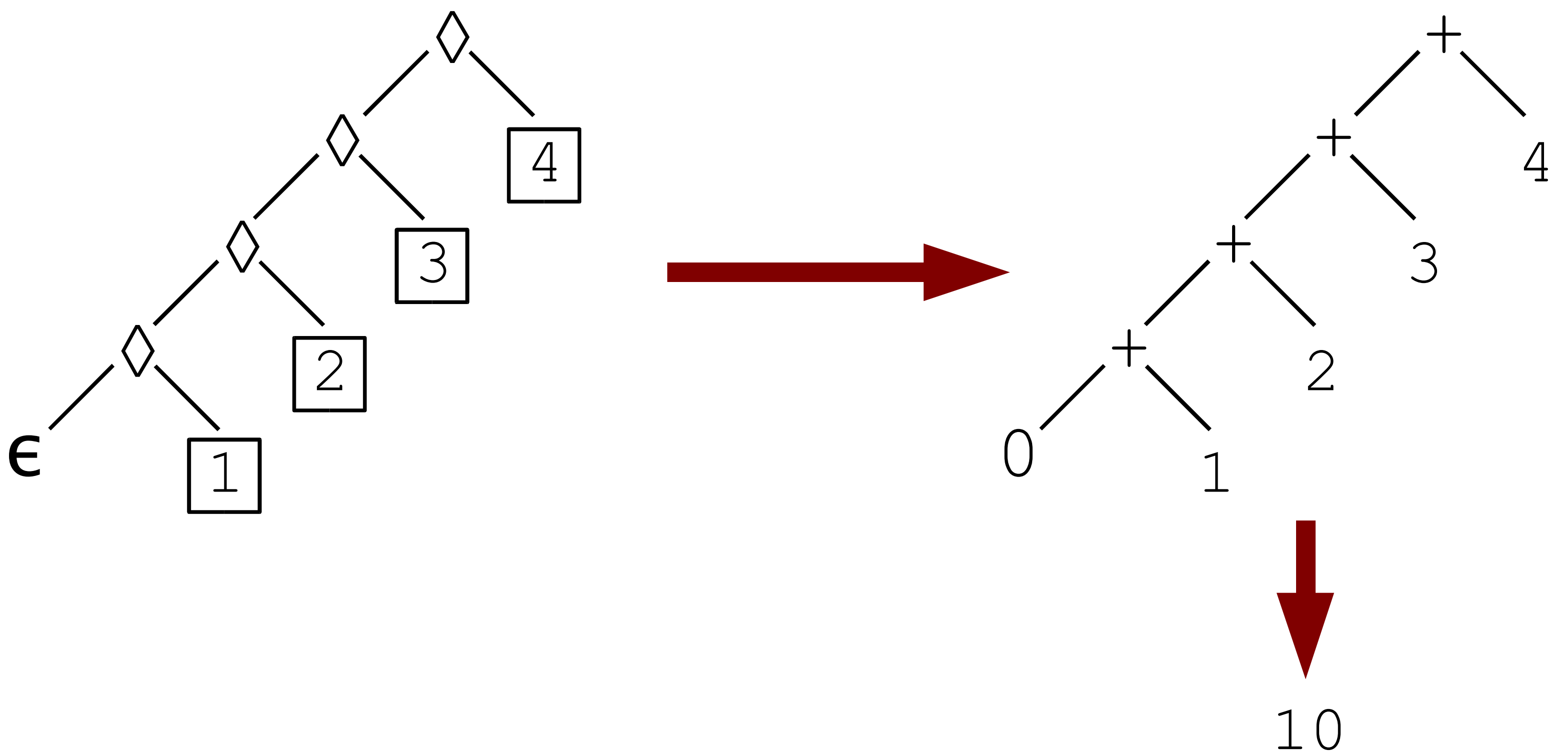
Possible Algebraic Properties of ◊

Associative	Commutative	Idempotent	
no	no	no	leaf trees
no	no	yes	BDD-like
no	yes	no	mobiles
no	yes	yes	weird
yes	no	no	lists
yes	no	yes	weird
yes	yes	no	multisets
yes	yes	yes	sets

The “Boom hierarchy”

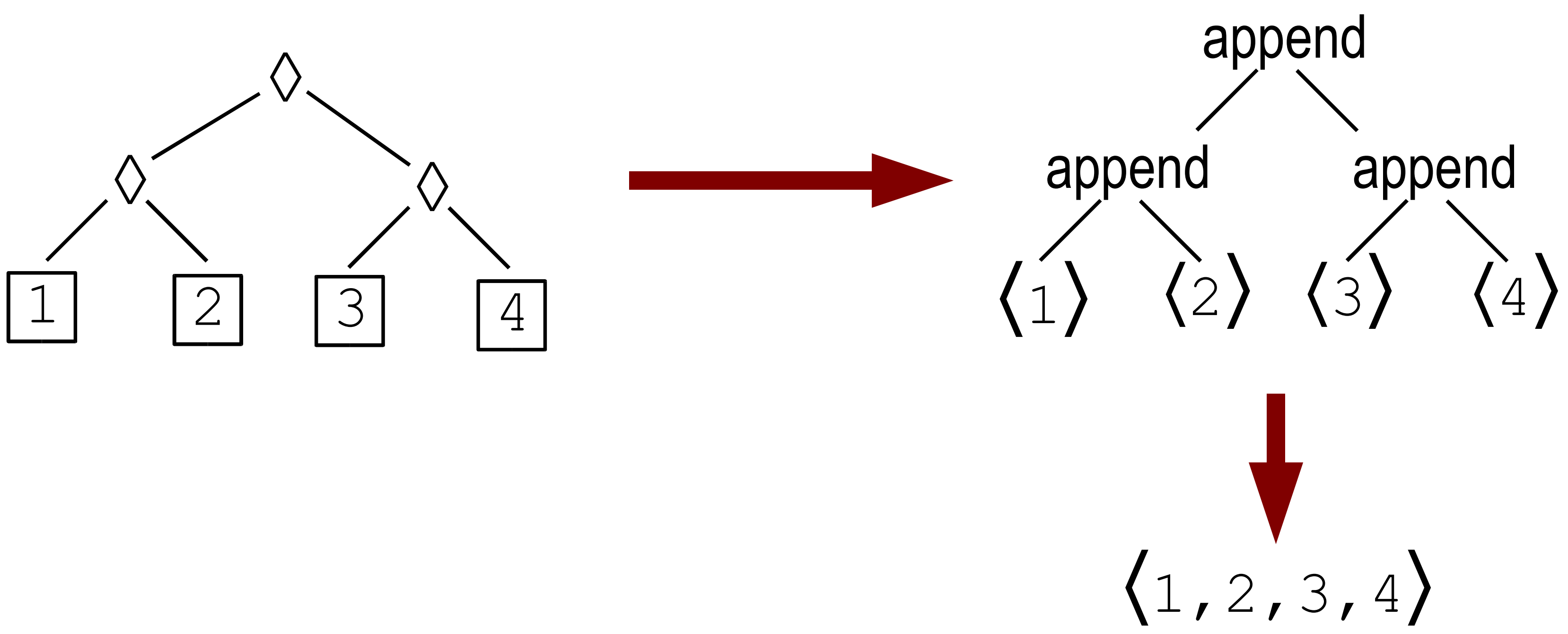
Catamorphism: Summation

Replace ◊ □ ϵ with + identity 0



Catamorphism: Lists

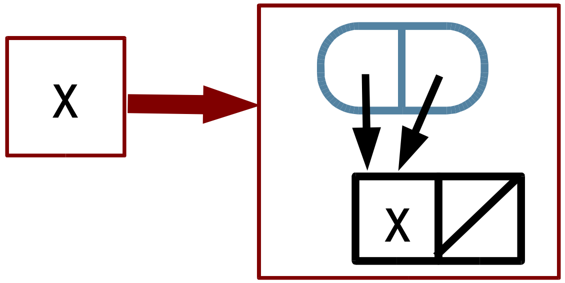
Replace $\lozenge$ $\square$ ϵ with append $\langle - \rangle$ $\langle\rangle$



Catamorphism: Splicing Linked Lists

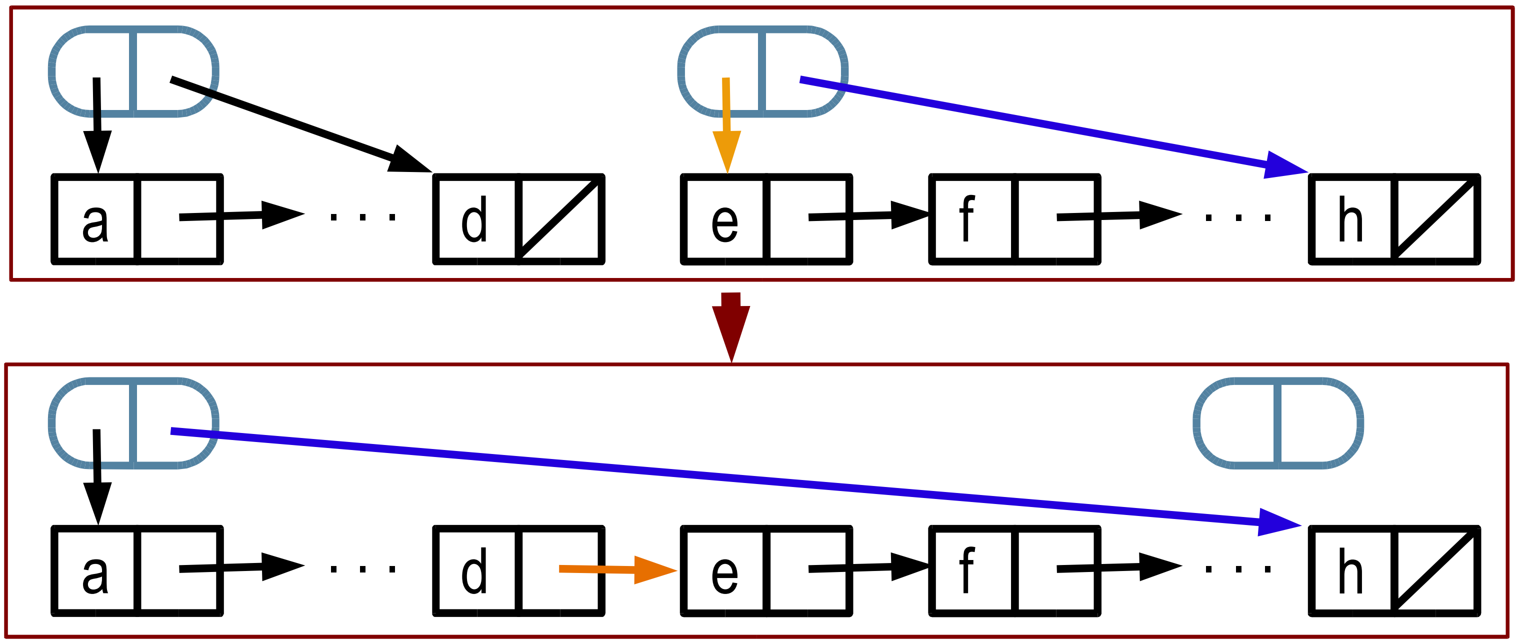
Replace ◊ □ ϵ with conc unitList nil

unitList:



(At the end, use the left-hand pointer of the final pair.)

conc:



Desugaring

`Σ[i←a,j←b,p,k←c] e`	becomes `Σ(f)`
`⟨ e | i←a,j←b,p,k←c ⟩`	becomes `makeList(f)`
for `i←a,j←b,p,k←c` **do** `e` **end**	becomes `forLoop(f)`

where `f =`

```
(fn (r)⇒
   (a).generate(r, fn (i)⇒
      (b).generate(r, fn (j)⇒
         (p).generate(r, fn ()⇒
            (c).generate(r, fn (k)⇒
               r.single(e))))))
```

Note: `generate` can be overloaded to exploit properties of r!

Implementation

```
opr Σ⟦T⟧(f: Generator⟦T⟧): T
    where { T extends Monoid⟦T,+,zero⟧ } =
  f.run(Catamorphism(fn(x,y)⇒ x+y, id, 0))

makeList⟦T⟧(f: Generator⟦T⟧): List⟦T⟧ =
  f.run(Catamorphism(append, fn(x)⇒ ⟨x⟩, ⟨⟩))

makeList⟦T⟧(f: Generator⟦T⟧): List⟦T⟧ =
  f.run(Catamorphism(conc, unitList, nil)).
      first

forLoop(f: Generator⟦()⟧): () =
  f.run(Catamorphism(par, id, ()))
```

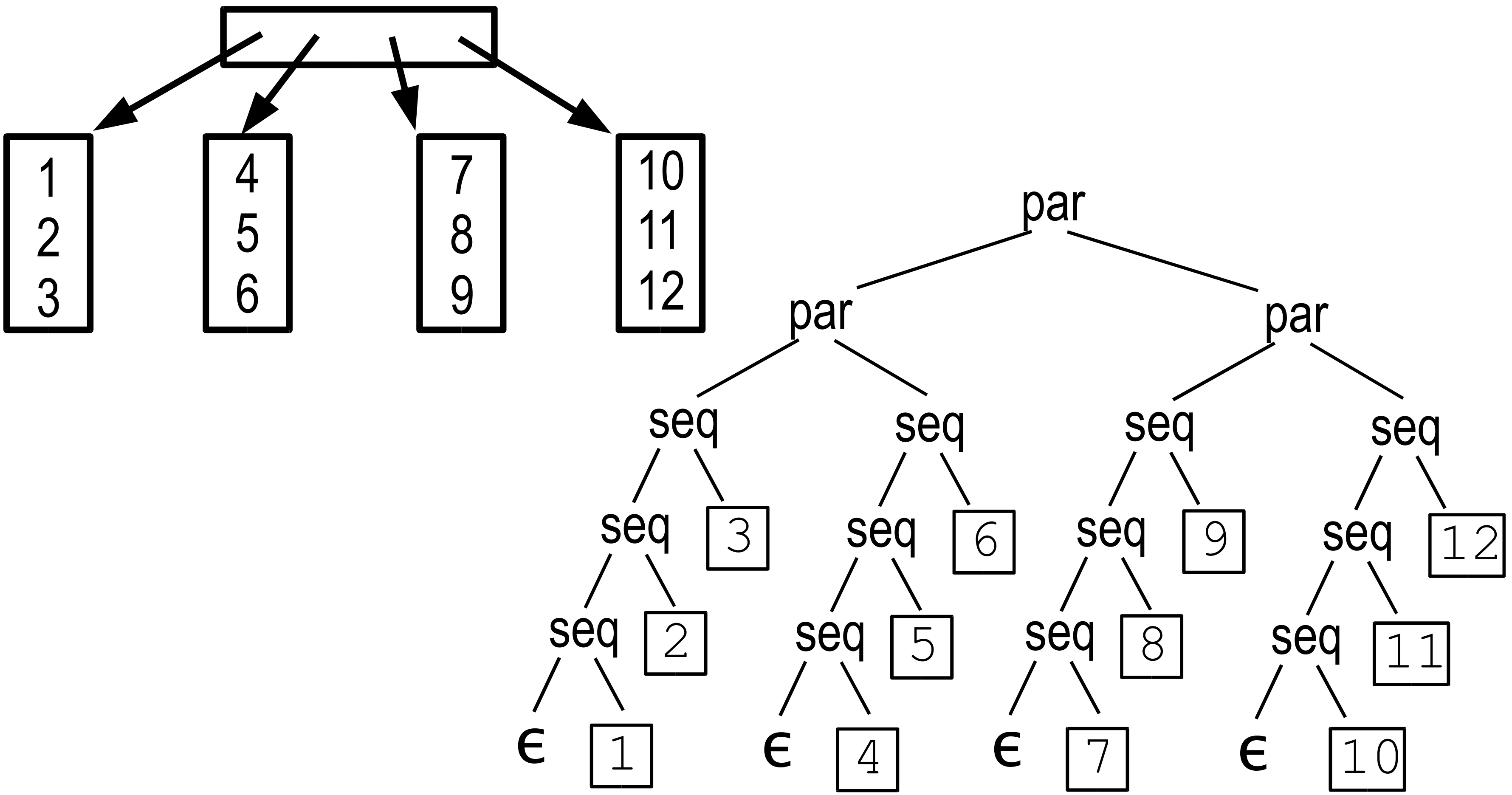
Implementation

```
value object Catamorphism⟦T,R⟧
   (join : (R,R)→ R,
    empty: R,
    single : T → R)
   map⟦U⟧(f : U → T) : Catamorphism⟦U,R⟧ =
      Catamorphism(join, empty, fn x ⇒ single(f(x)) )
end

trait Generator⟦T⟧ extends Object
  run⟦R⟧(c : Catamorphism⟦T,R⟧) : R =
        generate(c, fn x ⇒ x)
  generate⟦R⟧(c : Catamorphism⟦T,R⟧, f : T → R): R =
        run(c.map(f))
  size: ℤ64
end
```

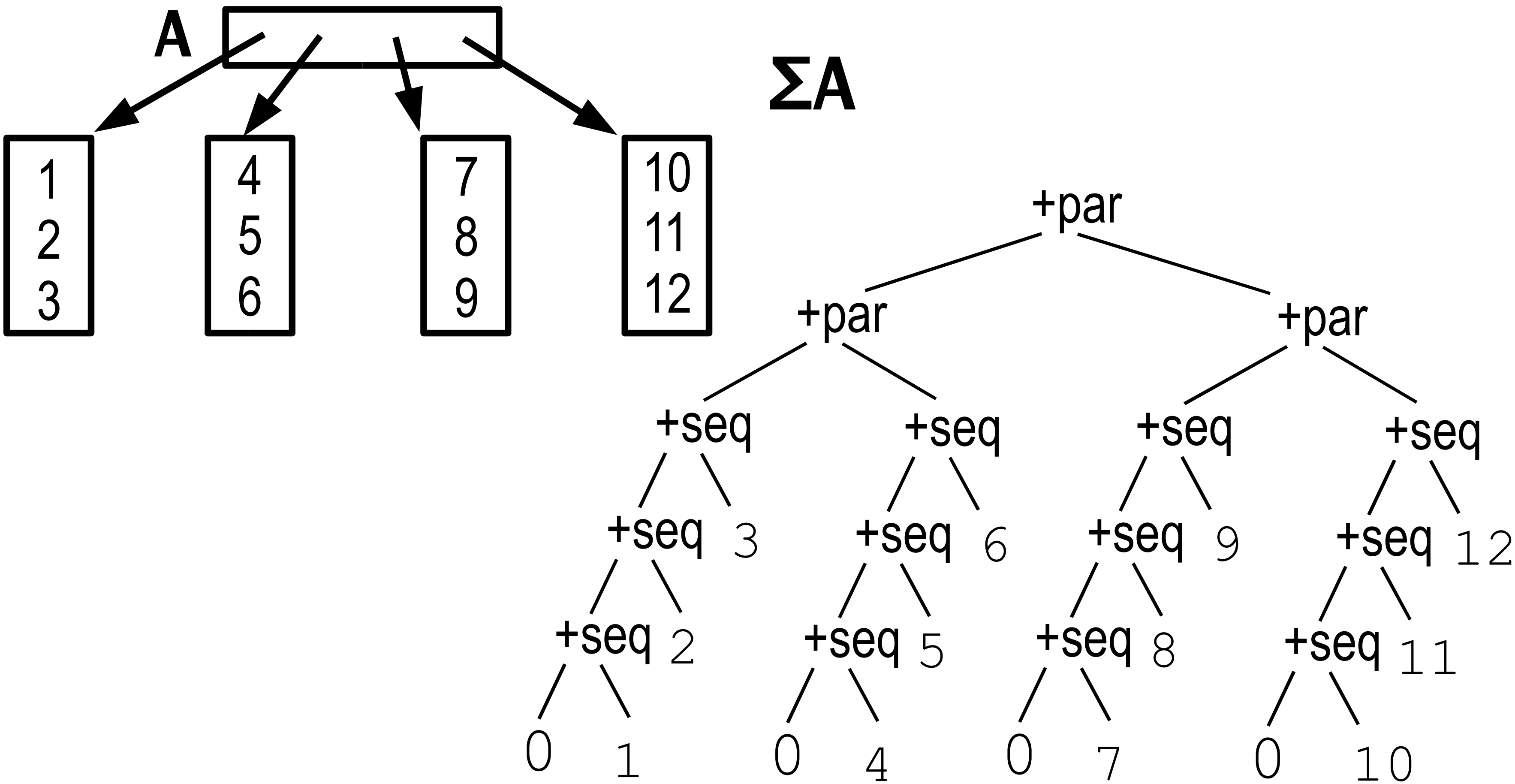
Generator Example

```
value object BlockedRange(lo: ℤ64, hi: ℤ64, b: ℤ64)
    extends Generator⟦ℤ64⟧
  size = hi - lo + 1
  run⟦R⟧(c : Catamorphism⟦ℤ64,R⟧) : R =
    if size ≤ max(b,1) then
      r : R = c.empty
      i : ℤ64 = lo
      while i ≤ hi do
        r := c.join(r,c.single(i))
        i += 1
      end
      r
    else
      mid = ⌊(lo + hi) / 2⌋
      c.join(BlockedRange(lo,mid,b).run(c),
             BlockedRange(mid+1,hi,b).run(c))
    end
end
```

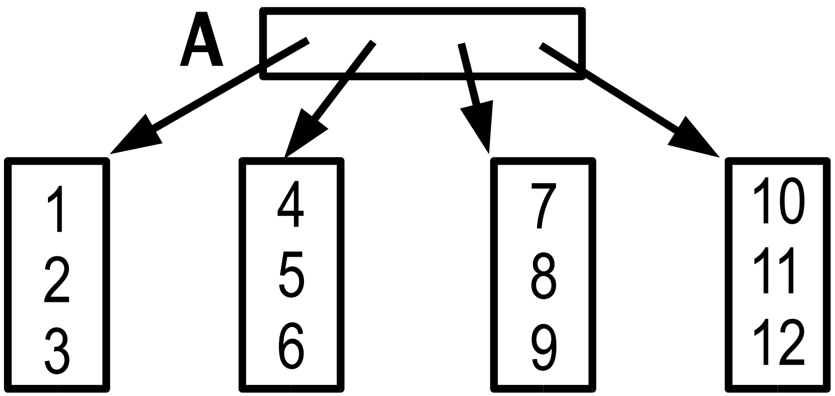
Generators Drive Parallelism



Generators Modify Reducers: Parallelism

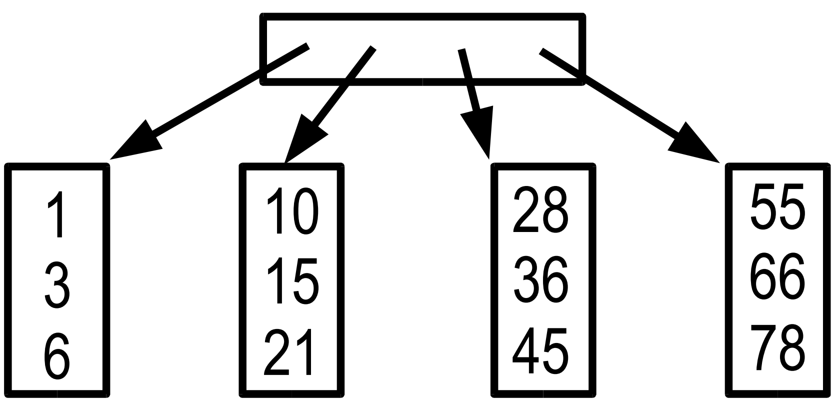


Generators Modify Reducers: Distribution



[x(x+1)/2 | x←A]

There are also ways (not shown here) for the programmer to specify a distribution explicitly.



Generalizing Comprehensions

- We can generalize the comprehension notation:

```
[ x ↦ y | x ← a.indices ][b.distribution]
⟨ f(x) | x ← xs ⟩^consume
```

- In full generality (using both features), we write:

```
⟨ e | g ⟩^consume[args]
```

- Comprehension yields generator G, called like so:

```
consume( G , args)
```

- Default subscript constructs list / array / set / map as appropriate

Summary: Parallelism in Fortress

- Regions describe machine resources.
- Distributions map aggregates onto regions.
- Aggregates used as generators drive parallelism.
- Algebraic properties drive implementation strategies.
- Algebraic properties are described by traits.
- Properties are verified by automated unit testing.
- Traits allow sharing of code, properties, and test data.
- Reducers and generators negotiate through overloaded method dispatch keyed by traits to achieve mix-and-match code selection.

Introduction
Language Overview
Basics of Parallelism

Components and APIs
Defining Mathematical Operators
Polymorphism and Type Inference

Parallelism: Generators and Reducers
Contracts, Properties, and Testing
Summary

Contracts

- Function contracts consist of three optional parts:
  - a **requires** part
  - an **ensures** part
  - an **invariant** part
- Example: **requires**

```
factorial(n:ℤ64) requires n≥0 =
  if n = 0 then 1
  else n factorial(n - 1)
  end
```

Contract Example

- Example: **ensures** and **invariant**

```
mangle(input:List) ensures sorted(result)
                      provided sorted(input)
                   invariant size(input) =
  if input ≠ Empty
  then mangle(first(input))
       mangle(rest(input))
  end
```

Properties

- Properties can be declared in trait declarations.
- Such properties are expected to hold for all instances of the trait and for all bindings of the property's parameters.
- Example:

```
trait Symmetric⟦T extends Symmetric⟦T,~⟧,
                         opr ~⟧
        extends BinaryPredicate⟦T,~⟧
  property ∀(a:T,b:T)(a~b) ↔ (b~a)
end
```

Tests

- Test functions are evaluated with every permutation of test data.
- If a non-test code refers to any part of test code, a static error is signaled.
- Example:

```
test s:Set⟦ℤ64⟧ =
  {-2000,0,1,7,42,59,1000,5697}

test fIsMonotonic[x←s,y←s] =
  assert(x ≤ y → f x ≤ f y)
```

Properties and Tests

- Properties are verified by automated unit testing.
- Properties can be named as property functions and can be referred to in a program's test code.
- If the result of a property function call is not true, a test failure is signaled.

Inheritance of Properties and Tests

- Traits allow sharing of code, properties, and test data.
- Algebraic constraints are described by traits.
- Multiple inheritance of contracts, properties, and tests of algebraic constraints are provided.

Algebraic Constraints Example

$\texttt{trait}\ \mathrm{BinaryPredicate}[\![T\ \texttt{extends}\ \mathrm{BinaryPredicate}[\![T, \sim]\!], \texttt{opr}\ \sim]\!]$
$\quad \texttt{opr}\ \sim(\texttt{self}, \mathit{other}\colon T)\colon \mathrm{Boolean}$
$\texttt{end}$

$\texttt{trait}\ \mathrm{Symmetric}[\![T\ \texttt{extends}\ \mathrm{Symmetric}[\![T, \sim]\!], \texttt{opr}\ \sim]\!]$
$\qquad \texttt{extends}\ \{\, \mathrm{BinaryPredicate}[\![T, \sim]\!]\,\}$
$\quad \texttt{property}\ \forall (a\colon T, b\colon T)\ (a \sim b) \leftrightarrow (b \sim a)$
$\texttt{end}$

$\texttt{trait}\ \mathrm{EquivalenceRelation}[\![T\ \texttt{extends}\ \mathrm{EquivalenceRelation}[\![T, \sim]\!], \texttt{opr}\ \sim]\!]$
$\qquad \texttt{extends}\ \{\, \mathrm{Reflexive}[\![T, \sim]\!], \mathrm{Symmetric}[\![T, \sim]\!], \mathrm{Transitive}[\![T, \sim]\!]\,\}$
$\texttt{end}$

$\texttt{trait}\ \mathrm{Integer}\ \texttt{extends}\ \{\, \mathrm{CommutativeRing}[\![\mathrm{Integer}, +, -, \cdot, \mathit{zero}, \mathit{one}]\!],$
$\qquad \mathrm{TotalOrderOperators}[\![\mathrm{Integer}, <, \leq, \geq, >, \texttt{CMP}]\!],$
$\qquad \ldots\}$
$\quad \ldots$
$\texttt{end}$

(This is actual Fortress library code.)

Example: Lexicographic Comparison

- Assume a binary `CMP` operator that returns one of Less, Equal, or Greater
- Now consider the binary operator `LEXICO`:

`LEXICO`	Less	Equal	Greater
Less	Less	Less	Less
Equal	Less	Equal	Greater
Greater	Greater	Greater	Greater

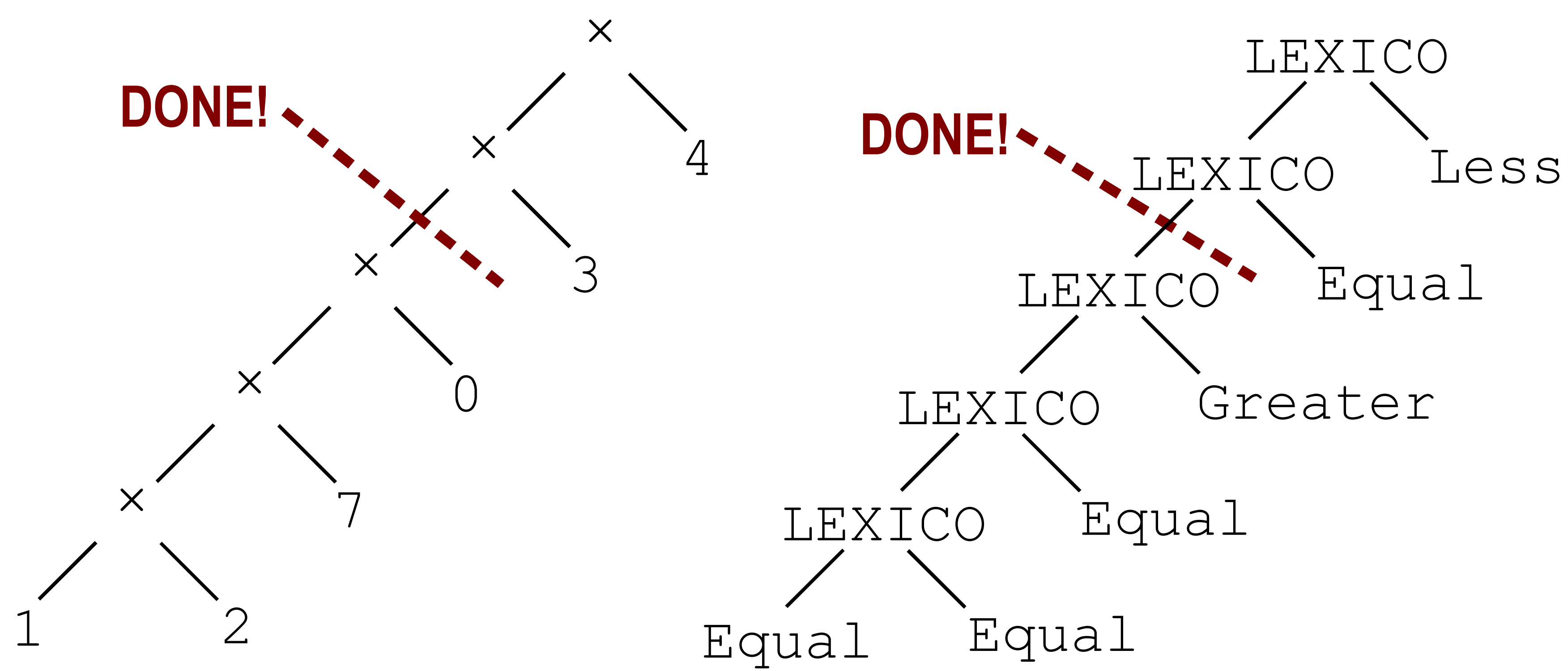
  > Associative (but *not* commutative)
  > Equal is the identity
  > Less and Greater are left zeroes

Algebraic Properties of LEXICO

```
trait Comparison extends {
    IdentityEquality⟦Comparison⟧,
    Associative⟦Comparison,LEXICO⟧,
    HasIdentity⟦Comparison,LEXICO,Equal⟧,
    HasLeftZeroes⟦Comparison,LEXICO⟧
  }
 ...
  test data = { Less, Equal, Greater }
end
```

A generator that detects the LEXICO catamorphism (rather, the fact that it has left zeros) can choose to generate special code.

Zeroes Can Stop Iteration Early



Code for Lexicographic Comparison

```
trait LexOrder⟦T,E⟧
  extends { TotalOrder⟦T,≤,CMP⟧,
            Indexable⟦LexOrder⟦T,E⟧,E⟧ }
  where { T extends LexOrder⟦T,E⟧,
          E extends TotalOrder⟦T,≤,CMP⟧ }

  opr =(self,other:T):Boolean =
    |self| = |other| AND:
      AND[i←self.indices] self[i]=other[i]

  opr CMP(self,other:T):Comparison = do
    prefix = self.indices ∩ other.indices
    (LEXICO[i←prefix] self[i] CMP other[i]) &
      LEXICO (|self| CMP |other|)
  end

  opr ≤(self,other:T):Boolean =
    (self CMP other) ≠ Greater

end
```

String Comparison

```
trait String
  extends { LexOrder⟦String,Character⟧, ... }
  opr [i:IndexInt]: Character = ...
  ...
  test data = { “foo”, “foobar”, “quux”, “” }
end
```

Introduction
Language Overview
Basics of Parallelism

Components and APIs
Defining Mathematical Operators
Polymorphism and Type Inference

Parallelism: Generators and Reducers
Contracts, Properties, and Testing
Summary

Fortress Goals

- Reduce application complexity
- Reduce compiler complexity
- Powerful language for library coding
  - > Put compiler complexity into modular Fortress source code
  - > Provide powerful abstractions for application coding
  - > Enable the language to grow
- Simplify application coding and deck checking
  - > Make mathematical code look like “whiteboard notation”
- Make it easy to code parallel algorithms

Guy Steele
Jan-Willem Maessen
http://research.sun.com/projects/plrg

Carl Eastlund, Guy Steele, Jan-Willem Maessen, Yossi Lev, Eric Allen, Joe Hallett, Sukyoung Ryu, Sam Tobin-Hochstadt, David Chase, João Dias